Instituto Tecnológico de Estudios Superiores de Monterrey

TE3001B.101

Fundamentación de Robótica

Challenge 2.

Gpo 101

Profesor:

Rigoberto Cerino Jiménez

Alumnos:

Daniela Berenice Hernández de Vicente	A01735346
Alejandro Armenta Arellano	A01734879
Dana Marian Rivera Oropeza	A00830027

Fecha: 27 de Febrero del 2023

Resumen

En este reporte se podrá observar una descripción de lo que son los custom messages, parameters y launch file, cómo funcionan, todo esto con tal de comprender más allá de simplemente resolver el challenge semanal impuesto por Manchester Robotics.
Aunado a estas descripciones y definiciones, se puede encontrar la solución paso por paso al challenge de esta segunda semana.

Objetivos

En este segundo challenge se espera que los estudiantes repasen los conceptos vistos en la sesión pasada.
Por lo cual la actividad consiste en lo siguiente:

- Crear un controlador para un sistema simple de primer orden en ROS. Este sistema representa el comportamiento dinámico de un motor DC.
- El nodo Sistema y un programa de simple estructura serán proporcionados por MCR2.
- El sistema de control puede ser "P", "PI", o "PID", otros controladores son aceptables sólo si el profesor lo ha autorizado previamente.

Objetivos específicos:

- Nodo System:
    - Es proporcionado por MCR2.
    - Las entradas y salidas del nodo son mensajes personalizados hechos específicamente para este sistema.
    - El estudiante deberá ocupar estos mensajes para comunicarse con el sistema.
    - Estos mensajes estarán en la carpeta "msg".
    - Los parámetro del sistema se pueden encontrar en "config/system_params.yaml".
- Nodo Controller:
    - Crear un nodo llamado "Controller" para generar una entrada de control al nodo system.
    - Este nodo debe publicar en el tópico "motor_input" y suscribirse a los tópicos "motor_output" y "set_point".
    - Los mensajes del controller "motor_input" y "motor_output" son los siguientes:

```
Motor input message (/motor_input):
float32 input          #Input to the system
float64 time           #Simulation time to be sent to the system

Motor output message (/motor_output):
float32 output         #Output of the system
float64 time           #Time stamp of the system
string status          #Status of the motor "Turning", "Not Turning", "Max Speed"
```

    - La salida del controller "motor_input" debe ser delimitado entre el intervalo -1 a 1 i.e., $u(k) \in [-1,1]$.
    - El mensaje para el tópico "set_point" debe ser definido por el estudiante.
        - Debe ser un mensaje personalizado.

        - Debe incluir al menos dos diferentes variables.
    - El nodo control, debe usar un archivo parámetro, para todas las variables de ajuste requeridas.
    - El tiempo de muestreo y la velocidad pueden ser los mismos que el nodo system.
    - Queda estrictamente prohibido utilizar alguna otra librería para Python además de NumPy. El controlador debe hacerse sin usar ningún controlador en línea predefinido.
- Launch File and Plotting:
        - Usar la herramienta "rqt_plot" para imprimir ambas señales.
        - Crear un archivo launch el cual debe poder acceder a las terminales requeridas para imprimir la información.
        - Debe ser capaz de abrir el "rqt_plot" e imprimir todas las señales requeridas en una misma ventana.
        - Debe ser capaz de cargar todos los archivos de parámetros automáticamente.

Introducción

Semanalmente se nos presentará un reto por Manchester Robotics, los cuales serán resueltos con la ayuda de programas especializados como ROS y ciertos lenguajes de programación como Python.

De igual manera cada semana contaremos con una clase base con respecto a lo que se manejara durante el challenge semanal, así como una sesión de preguntas y respuestas donde se buscarán resolver dudas con respecto al mismo challenge de la semana.

Por otro lado también se nos proporcionarán algunas actividades las cuales hasta cierto punto serán un pequeño desglose del challenge semanal, claro está que contarán con una menor dificultad.

En esta segunda semana al ya contar con una noción clara de la arquitectura de trabajo de ROS, se nos introdujo a los siguientes temas:

- ROS Namespaces:
    - Un namespace en ROS se puede visualizar como un directorio que contiene artículos con diferentes nombres.
    - Esos artículos pueden ser nodos, tópicos, o otros namespaces (dependiendo de la jerarquía).
    - Existen varias formas de definir los namespaces. La más fácil es por una línea de comando, la cual es muy sencilla, pero para proyectos largos no es recomendada.
- ROS Parameters:
    - Son variables con algunos valores predefinidos que se almacenan en archivos separados o codificados en algún programa tal que el usuario pueda acceder fácilmente para modificar el valor.
    - Al mismo tiempo, los parámetros se pueden compartir entre diferentes programas para evitar reescribirlos o recompilar los nodos (C++).
    - ROS usa parámetros para evitar crear dependencias o reescribir nodos.

    - ROS utiliza este “diccionario” para almacenar y compartir los parámetros para ser usados por sus nodos. Este “diccionario” es llamado “Parameter Server”.
- ROS Parameter Server:
    - Como se mencionó anteriormente ROS permite cargar variables en un servidor, ejecutado por el master el cual es accesible por todos los nodos del sistema.
    - Los nodos utilizan este servidor para almacenar y recuperar los parámetros al momento de ser ejecutados.
    - Esta es una librería estática vista globalmente.
    - Los parámetros están compuestos por un nombre y un tipo de dato, ROS puede usar los siguientes tipos de parámetros:
        - 32-bit integers.
        - Strings.
        - base64-encoded binary data.
        - Booleans.
        - iso8601 dates
        - Doubles.
        - Lists.
    - Los parámetros pueden ser globales con un namespace local o privados, específicos para un nodo.
    - Es útil señalar que el servidor de parámetros depende de la ejecución del master ROS no al tiempo de ejecución del nodo. Por lo tanto, si se mata un nodo pero no al master, el parámetro mantendrá el valor si fue modificado.
- ROS Parameter Files:
    - ROS ofrece la capacidad de definir parámetros mediante un parameter file.
    - Este tipo de archivos son archivos de configuración, escritos en YAML. Estos archivos son usados comúnmente en otros lenguajes para configurar parámetros o variables.
- ROS Custom Messages:
    - ROS tiene algunos mensajes predefinidos como std_messages, geometric_messages, etc.
    - A veces, se requiere que las estructuras del mensaje sean modificadas para una aplicación personalizada.
    - ROS permite al usuario personalizar mensajes y crear nuevos mensajes.
    - Los custom messages son una manera de personalizar tus propios mensajes para un motivo o aplicación en específico**.**
    - Los custom messages son creados por el usuario y deben estar vinculados al paquete donde se usarán.

Solución del problema

Los nodos algunos han sido creados y otros los ha proporcionado MCR2, de igual manera el controlador ha sido creado y obtenido mediante prueba y error.
Por otro lado, los comandos principales para el código han sido vistos en el challenge de la semana pasada, por lo cual en este los hemos omitido.

Aunado a esto, describimos los códigos implementados en la solución del challenge semanal.

Nodo Controller:

```
#!/usr/bin/env python
import rospy
import numpy as np
from std_msgs.msg import Float32
from pid_control.msg import set_point
from pid_control.msg import motor_input
from pid_control.msg import motor_output

kp = rospy.get_param("kp",8)
ki = rospy.get_param("ki",0.006)
tau = rospy.get_param("tau",6.28)
R = rospy.get_param("R",0.01)
dt = rospy.get_param("dt",0.2)


signal_data = 0.0
time_data = 0.0

output_data = 0.0
time_data2 = 0.0

error = 0.0
angularVelocity = 0.0

error_acumulado = 0.0
ultima_medicion = 0.0


msg = motor_input()
msg.time = 0.0
msg.input = 0.0
```

```
def callback(msg):
    global signal_data, time_data
    signal_data = msg.signal_y
    time_data = msg.time_x

def callback2(msg):
    global output_data, time_data2
    output_data = msg.output
    time_data2 = msg.time

def stop():
 #Setup the stop message (can be the same as the control message)
  print("Stopping")
```

```python
if __name__=='__main__':
    #Initialise and Setup node
    pub=rospy.Publisher("motor_input",motor_input, queue_size=10)
    pub_controlador=rospy.Publisher("control",Float32, queue_size=10)
    rospy.init_node("controller")
    rospy.Subscriber(rospy.get_namespace()+"set_point", set_point, callback)
    rospy.Subscriber(rospy.get_namespace()+"motor_output", motor_output, callback2)
    rate = rospy.Rate(10)
    rospy.on_shutdown(stop)

    while not rospy.is_shutdown():

        error = 0.0 + signal_data
        error_acumulado += error * dt  ## multiplicar por "dt"
        accion_proporcional = kp * error
        accion_integral = ki * error_acumulado
        accion_control = accion_proporcional + accion_integral
        u = accion_control * R
        velocidad = ultima_medicion + ((u - ultima_medicion) / tau)
        #pub.publish(Float32(velocidad))
        ultima_medicion = velocidad
        signal=ultima_medicion

        msg.time = time_data
        msg.input = signal
        pub.publish(msg)
        pub_controlador.publish(signal)
        print_info = "%3f | %3f" %(signal,time_data)
        rospy.loginfo(print_info)
        rate.sleep()
```

En este código se crea un control en la entrada del nodo sistema. De igual manera el nodo pública en el tópico “motor_input” y se suscribe a los tópicos “motor_output” y “set_point”. La salida del controller “motor_input” está delimitado entre el intervalo -1 a 1 i.e., $u(k) \in [-1,1]$ y se mandan los mensajes personalizados por nosotros.

Nodo Set Point Generator:

```
#!/usr/bin/env python
import rospy
import numpy as np
from pid_control.msg import set_point
from std_msgs.msg import Float32

msg = set_point()
msg.time_x = 0.0
msg.signal_y = 0.0

if __name__=='__main__':
    pub_signal=rospy.Publisher("set_point",set_point, queue_size=10)
    pub_motor=rospy.Publisher("signal",Float32, queue_size=10)
    rospy.init_node("Set_Point_Generator")
    rate = rospy.Rate(10)
    t0= rospy.Time.now().to_sec()

    while not rospy.is_shutdown():
        P=rospy.get_param("P",21)
        phase= rospy.get_param("phase",0)
        amplitud= rospy.get_param("amplitud",1)
        offset= rospy.get_param("offset",0)
        w=2*np.pi/P
        t= rospy.Time.now().to_sec()-t0
        timeset=t
        signal=(np.sin(w*t+phase)*amplitud)+offset

        msg.time_x = t
        msg.signal_y = signal
        pub_signal.publish(msg)
        pub_motor.publish(signal)
        print_info = "%3f | %3f" %(signal,t)
        rospy.loginfo(print_info)

        rate.sleep()
```

En este código se utiliza el código creado en el challenge 1, el cual pública en el tópico “Set_point” una señal y un mensaje.

Nodo System:

```
#!/usr/bin/env python
import rospy
import numpy as np
from pid_control.msg import motor_output
from pid_control.msg import motor_input
from std_msgs.msg import Float32

class SimpleSystem:

  def __init__(self):

    #Set the parameters of the system
    self.sample_time = rospy.get_param("/system_sample_time",0.02)
    self.max_speed = rospy.get_param("/system_max_speed",13.0)
    self.min_input = rospy.get_param("/system_min_input",0.2)
    self.param_K = rospy.get_param("/system_param_K",13.2)
    self.param_T = rospy.get_param("/system_param_T",0.05)
    self.init_conditions = rospy.get_param("/system_initial_cond",0)

    # Setup Variables to be used
    self.first = True
    self.start_time = 0.0
    self.current_time = 0.0
    self.last_time = 0.0
    self.proc_output = 0.0

    # Declare the input Message
    self.Input = motor_input()
    self.Input.input = 0.0
    self.Input.time = 0.0


    # Declare the  process output message
    self.output = motor_output()
    self.output.output= self.init_conditions
    self.output.time = rospy.get_time()
    self.MotorStatus(self.init_conditions)
```

```
  # Setup the Subscribers
  rospy.Subscriber("/motor_input",motor_input,self.input_callback)

  #Setup de publishers
  self.state_pub = rospy.Publisher("/motor_output", motor_output, queue_size=1)

#Define the callback functions
def input_callback(self,msg):
  self.Input = msg

#Define the main RUN function
def run (self):
  #Variable setup
  if self.first == True:
    self.start_time = rospy.get_time()
    self.last_time = rospy.get_time()
    self.current_time = rospy.get_time()
    self.first = False
#System
  else:
    #Define sampling time
    self.current_time = rospy.get_time()
    dt = self.current_time - self.last_time

    #Dynamical System Simulation
    if dt >= self.sample_time:
      #Dead-Zone
      if(abs(self.Input.input)<=self.min_input):
        self.proc_output+= (-1.0/self.param_T * self.proc_output + self.param_K/self.param_T * 0.0) * dt
      #Saturation
      elif (((-1.0/self.param_T * self.proc_output + self.param_K/self.param_T * self.Input.input)>0.0 and self.proc_output> self.max_speed)or ((-1.0/self.param_T * self.proc
        self.proc_output+= (-1.0/self.param_T * self.proc_output + self.param_K/self.param_T * ((1/self.param_K)*self.proc_output)) * dt
      #Dynamic System
      else:
        self.proc_output += dt*((-1.0/self.param_T) * self.proc_output + (self.param_K/self.param_T) * self.Input.input)
```

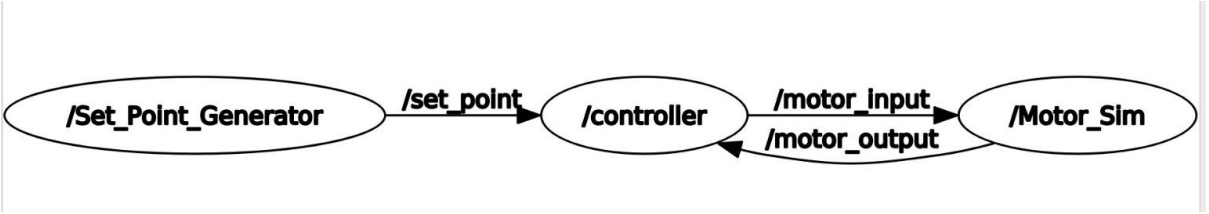
```
                #Message to publish
                self.output.output= self.proc_output
                self.output.time = rospy.get_time() - self.start_time
                self.MotorStatus(self.proc_output)
                #Publish message
                self.state_pub.publish(self.output)

                self.last_time = rospy.get_time()

              #else:
              #self.state_pub.publish(self.output)

            # Motor Status Function
            def MotorStatus(self,speed):
              if (abs(speed)<=abs(self.param_K*self.Input.input*0.8) and abs(self.Input.input)<=self.min_input):
                self.output.status = "Motor Not Turning"
              elif (abs(speed)>=self.max_speed):
                self.output.status = "Motor Max Speed"
              else:
                self.output.status = "Motor Turning"



           #Stop Condition
            def stop(self):
            #Setup the stop message (can be the same as the control message)
              print("Stopping")
              self.output.output= 0.0
              self.output.time = rospy.get_time() - self.start_time
              self.output.status = "Motor Not Turning"
              self.state_pub.publish(self.output)
              total_time = rospy.get_time()-self.start_time
              rospy.loginfo("Total Simulation Time = %f" % total_time)
```

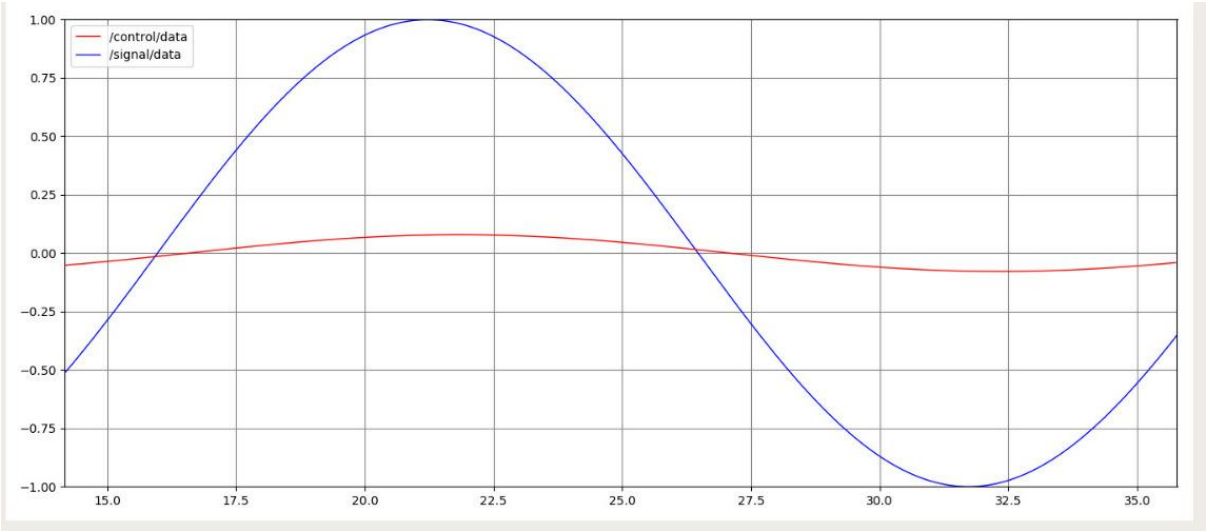
```
    self.output.output= 0.0
    self.output.time = rospy.get_time() - self.start_time
    self.output.status = "Motor Not Turning"
    self.state_pub.publish(self.output)
    total_time = rospy.get_time()-self.start_time
    rospy.loginfo("Total Simulation Time = %f" % total_time)


if __name__=='__main__':

 #Initialise and Setup node
 rospy.init_node("Motor_Sim")
 System = SimpleSystem()

 # Configure the Node
 loop_rate = rospy.Rate(rospy.get_param("/system_node_rate",100))
 rospy.on_shutdown(System.stop)

 print("The Motor is Running")
 try:
  #Run the node
  while not rospy.is_shutdown():
   System.run()
   loop_rate.sleep()

 except rospy.ROSInterruptException:
```

En este código se utiliza el node proporcionado por MCR2 que simula el sistema de primer orden Y(s)/U(s) =K/(Ts+1) aproximando de esta forma un simple motor DC con su encoder y driver.
La entrada y salida del nodo son mensajes personalizados que han sido creados específicamente para el sistema.

Resultados

En este apartado se muestra la relación que existe entre los canales de comunicación y entre los nodos para la transmisión de la señal de entrada y salida, y su procesamiento.
Todo esto gracias a que el nodo set_point_generator manda la información a través de la señal set_point el cual llega al segundo nodo “controller” donde la información se ve modificada por el controlador previamente establecido, posteriormente se manda la información mediante la señal llamada “motor_input”, la cual llega al tercer nodo “system”, siendo este el que regrese la información hacia el nodo “controller” mediante la señal “motor_output”, todo esto con tal de que la información pueda seguir siendo modificada y así generar un ciclo.





Conclusiones

Aún cuando en algunos momentos parecía inalcanzable, se consiguió realizar con éxito cada uno de los objetivos antes mencionados.

Se logró obtener la creación de un controlador para un sistema simple de primer orden en ROS, que representará el comportamiento dinámico de un motor DC.

Se logró obtener el diagrama y la señal esperada por MCR2.